CS4100/5100 COMPILER DESIGN PROJECT
Code Generation in Part 4: #2 CONDITIONALS AND DOWHILE
Fall 2022

The implementation of code generation for the *selection (IF)* and *iteration (DOWHILE)* structures will rely on a standard pattern for handling the conditional part of the construct. Note the similarity in handling the conditional:

1. DOWHILE <condition>   {NOTE: The FOR loop is a specialized WHILE loop}
    a. Evaluate the WHILE condition
    b. If the condition is FALSE, jump past the bottom of the WHILE loop
    c. Else continue executing into the loop body
2. IF <condition>
    a. Evaluate the IF condition
    b. If the condition is FALSE, jump past the end of the if, to the ELSE if it is there
    c. Else continue executing into the IF body, and at the end, jump past the ELSE

Consistently, in each case above, the condition is evaluated, and a jump (= a Quad conditional BRANCH instruction) occurs when the condition is false.

Each <condition> is represented in the CFG as the “relational expression”, abbreviated as <relexpression>.

```
<relexpression> -> <simple expression>  <relop>  <simple expression>
<relop>        ->   $EQ | $LSS | $GTR | $NEQ | $LEQ | $GEQ
```

If the action taken for evaluating all relational expressions is standardized as using the result of subtracting the right side operand of the relational operator from the left side operand, i.e. for

```
          A < B
```

generating a Quad to perform the operation:

```
          Temp = A - B
```

Then the Quad generated will always be:

```
          SUB, indexof(A), indexof(B), indexof(Temp)
```

This makes a simple, consistent pattern for use in all conditional statements. Consider the relationship between the relational operators and the needed Quad BRANCH instruction opcodes. The result of subtracting *B* from *A* will be placed into *Temp*, which can be tested by the branching opcodes for positive (>0), negative (<0), zero (=0), not positive (<=0) not negative (>=0), and not 0 (<>0). The following table summarizes this:

A RELOP B	A - B RESULT	TRUE BRANCH	FALSE BRANCH
A = B	0	BZ	BNZ
A <> B	NOT 0	BNZ	BZ
A < B	< 0	BN	BNN
A > B	> 0	BP	BNP
A <= B	<= 0	BNP	BP
A >= B	>= 0	BNN	BN

This implies a simple function to convert a given relational operator into its corresponding FALSE BRANCH opcode in order to facilitate Quad creation:

```
int relopToOpcode(int relop);
     {int result;
      switch relop:
          {
          EQUAL: result = BN_ZOPCODE; break;
          NOTEQUAL: result = BZ_OPCODE; break;
          LESS: result = BNN_OPCODE; break;
          //etc.
          }
      return result;
     }
```

IMPORTANT DEVIATION FROM NON-TERMINAL CONVENTION: First, the <relexpression> function can be implemented using the above function. Because it will generate the actual Quads needed by its calling function, relexpression will break with the convention used in most other non-terminal functions which return the Symbol Table index of an operand. Instead, it will return the Quad Table index of the Branch instruction which needs to be filled in by the calling function. This will be illustrated:

```
int relexpression;
     {int left, right, saveRelop, result, temp;
      left = simpleexpression; //get the left operand, our ‘A’
      saveRelop = relop;          //returns tokenCode of rel operator
      right = simpleexpression;//right operand, our ‘B’
      temp = GenSymbol;        //Create temp var in symbol table
      Quads.addQuad(SUB_OPCODE, left, right, temp); //compare
      result = nextQuad;       //Save Q index where branch will be
      Quads.addQuad(relopToOpcode(saveRelop),temp,0,0);//target set
later
      return(result);
     }
```

Relexpression creates the compare and branching quads related to it, and by design, it will correctly generate exactly what is needed for all of the program control structures defined in the P22 language.

The DoWhile Loop
Code for the DoWhile loop consists of the following required elements:

1) A condition check
2) A conditional branch around the loop body if the condition is FALSE
3) An unconditional branch from the bottom of the loop body to the start of the condition check at the top of the loop.

Here is an approach when the **dowhile** token is found in the <statement> switch structure:

```
... inside of statement....
else .....
  if (tokenCode == DOWHILE)
   {
    // declare above int saveTop, branchQuad
    GNT;                                //move past this token
    saveTop = nextQuad; //Before generating code, save top of loop
                        // where unconditional branch will jump
    branchQuad = relexpression; //tells where branchTarget to be set
    If (tokenCode == DO)        //move past DO
     {
      GNT;
      statement;                //the loop body is processed
      AddQuad(Branch_Op, 0, 0, saveTop);//jump to top of loop
                                        //backfill the forward branch
      //Quad function for ease- set 3rd op
      Quad.setQuadOp3(branchQuad,nextQuad);//conditional jumps nextQuad
     }
    ELSE ... {Handle missing DO error}
   } //end of while structure
  else ....
```

The If/Else Statement
The If, with optional Else, is a bit more complicated to follow than the While, because there are a couple of unknown branch targets to be backfilled when they become known. The basic elements to be built are:

1) A condition check
2) A conditional branch around the IF body if the condition is FALSE
3) If an ELSE is found, add an unconditional branch around the ELSE body at the end of the IF body

Here is an approach when the **if** token is found in the <statement> switch structure:

```
        //variables needed:
        //      int branchQuad, patchElse
else .....
  if (tokenCode == IF)
   {
    GNT;                        // move past ‘if’
    branchQuad = relexpression; //tells where branchTarget to be set
                                // to jump around TRUE part
```

```
 if tokencode == THEN        //all ok, continue
  {
   GNT;                      // move past ‘then’
   statement;                //all if body quads are genned
   if tokencode == ELSE      //have to jump around to ??
    {
     GNT;                    // move past ELSE
     patchElse = nextQuad;   //save backfill quad to jump around
                             // ELSE body, target is unknown now
     AddQuad(Branch_op, 0, 0, 0);
                             //backfill the FALSE IF branch jump
     Quad.setQuadOp3(branchQuad,nextQuad);//conditional jump
     statement;              // gen ELSE body quads
                             // fill in end of ELSE part
     Quad.setQuadOp3(patchElse, nextQuad);
    }
   else                      //no ELSE encountered, fix IF branch
     Quad.setQuadOp3(branchQuad, nextQuad);
  }                          //if the THEN was found
 else                        // error, no THEN
     error(...);
}                            // end of IF statement stuff
```