CS39003 Compilers Laboratory
Autumn 2026
Assignment 10
Date of posting: 27-Oct-2026

Intermediate code generation: Branching and looping

This assignment is a continuation of the previous two assignments (LA8 and LA9). The MathC programs that you wrote for LA8 and LA9 should continue to generate 3-address codes exactly as before. Now, we introduce general branching and looping constructs to the language.

Assume that the standard relation (comparison) operators **==**, **!=**, **<**, **<=**, **>**, and **>=** apply to all of the four basic types `int`, `rtl`, `flt`, and `cpx` with the type hierarchy `int` < `rtl` < `flt` < `cpx`. Like binary operations, the two expressions being compared must be of the same type. If not, type widening is applied to the expression with the smaller type. Moreover, the type prefix should appear in the comparison Operation. Examples: `jmpieq`, `jmpfge`, `jmprne`, `jmpclt`.

Part 1: Branching

We use the same grammar for assignments as in LA9. In particular, the productions under ASGN will continue to work as they did in LA9. You only need to change the type of ASGN to `nextlist` (it is empty for assignment statements). The productions for the start symbol PROG will now change in order to accommodate other types of statements.

```
PROG    →   LIST

BLOCK   →   { LIST }

LIST    →   LIST STMT | ϵ

STMT    →   ASGN
STMT    →   if ( BOOL ) BLOCK
STMT    →   if ( BOOL ) BLOCK else BLOCK
STMT    →   for ( ID = EXPR to EXPR step EXPR ) BLOCK
STMT    →   for ( ID = EXPR downto EXPR step EXPR ) BLOCK
```

In Part 1, you handle only the if and the if-else statements. The for loops are to be handled in Part 2. The above grammar allows empty lists inside programs and blocks. In order to avoid the dangling-else ambiguity, every block in a branching or looping statement (even if it consists of a single statement or none at all) should appear between matching braces. The productions for boolean expressions are given below.

```
BOOL    →   EXPR RELOP EXPR
BOOL    →   BOOL || BOOL
BOOL    →   BOOL && BOOL
BOOL    →   ! BOOL
BOOL    →   ( BOOL )

RELOP   →   == | != | < | <= | > | >=
```

Like LA9, you continue to store the 3-address instructions in a table QT of quads. You generate 3-address instructions on the fly, so you use **backpatching** for filling out the empty jump labels. For boolean expressions and branching statements, use the SDT as given in Section 6.7 of the Dragon Book. Use the markers M and N as explained in the book. There is one problem with this SDT, namely it does not specify how to backpatch the last statement in LIST. This can be handled at the parent of LIST (when no more statements are available for adding to LIST). More specifically, during the reductions PROG → LIST and BLOCK → **{** LIST **}**, the `nextlist` of LIST should be backpatched by the next available quad number.

A boolean expression has two synthesized attributes `truelist` and `falselist`. A statement has a single synthesized attribute `nextlist`. Each such list should store a set of quad numbers (integers). The list is not needed to be ordered, because it does not matter in which order the quads in a list receive the missing jump label. Write a separate source file to implement list-related utilities: `makelist()`, `merge()`, and `backpatch()` (as explained in the Dragon Book). The

yacc `%type` of STMT should be a reference (a pointer) to a list. The yacc `%type` of BOOL can be a pointer to a structure consisting of two list references. We leave the details for the implementation of lists to you. But note that you should have a C-compatible implementation because list references will be in the `%union` directive of yacc.

The grammar for boolean expressions as given above is ambiguous. For example, `B1 || B2 && B3` may be parsed as `B1 || (B2 && B3)` or as `(B1 || B2) && B3`. Assume that the boolean operators `||` and `&&` are left-to-right associative, and the negation operator `!` is non-associative. In terms of precedence, the operators satisfy `||` < `&&` < `!`. Set these associativity and precedence constraints using yacc directives (instead of altering the grammar).

Club together **all** your source files in a single tar/tgz/zip archive, and submit that file. The archive must also include a *makefile*, and the source files (updated or in the original form as the case is) used in LA8 and LA9. Do not submit only the new files.

Part 2: Looping

Implement MathC for loops in this part. The Dragon book handles only while loops. So the SDT for these for loops needs to be specifically designed. The two for loops can be rewritten as while loops as follows. Based on these, design your own translation scheme. Introduce additional markers and jump labels as necessary.

for (`var` = `start` to `last` step `amt`) { `body` }

```
var = start;
while (var <= last) {
    body
    var = var + amt;
}
```

for (`var` = `start` downto `last` step `amt`) { `body` }

```
var = start;
while (var >= last) {
    body
    var = var - amt;
}
```

The loop variable `var` is given the type of `start` by the initializing assignment. The type of `var` will be used in all loop-control operations. In the comparison, `last` will be typecast to the type of `var` (if necessary, even if it leads to narrowing). Likewise, for the increment or decrement of `var`, `amt` should be first typecast to the type of `var`, and then the addition or subtraction is carried out. The statements in `body` follow the type constraints as in LA8 and LA9. The programmer may alter the values of `var` and the variables involved in `last` and `amt`. However, assume that MathC programmers do not alter types of variables in loops (your program does not need to enforce this restriction). In order to see why, look at the note on the last page (after the Sample Output).

The grammar states that `start`, `last`, and `amt` are expressions (not necessarily constants). Codes for all these three expressions are generated before the code for the loop `body`. In fact, the generated code will look as follows.

Code for `var = start;`
Code for `last` (if any)
Code for conditional jumps based on the comparison of `var` with `last`
Code for `amt` (if any)
Code for incrementing/decrementing `var` by `amt`
Code for jump to update `last` and comparing with `var`
Code for `body`
Code for jump to update `amt` (and subsequently `var`)

The `body` may change the variables used in `last` and/or `amt`. Therefore `last` and `amt` must be reevaluated before the loop condition is checked after each iteration of the loop. For example, the following two loops are equivalent.

```
for (i = 0 to n - 1 step 1) {
    body
}
```

```
j = n + 1;
for (i = 0 to j - 2 step 2) {
    body with i replaced by i / 2
    j = j + 1;
}
```

The second loop changes `j`, so `last` (that is, `j - 2`) should be reevaluated before comparing with the current `i`.

Finally, `amt` should be positive (after typecasting to the type of `var`). Since `amt` is an expression, it cannot be evaluated during compilation. If `amt` evaluates to a value ≤ 0 during run time, its MathC program may enter an infinite loop, but that is not your fault (first-year PDS students will always blame the compiler and the compiler designers anyway).

Again club together **all** your source files (and the *makefile*) in a single tar/tgz/zip archive, and submit that file.

Sample Output

We work with the following input file.

```
x = 2000 // 7;
y = 3000 // 8;

if (x < 100 || x > 200 && x != y) { z = 0; }
if ((x < 100 || x > 200) && x != y) { z = 1; }

n = 100;
j = n + 1;
for (i = 0 to j - 2 step 2.0) {
   if (i == 0) {
      A[i] = 1.;
   } else {
      k = i / 2;
      A[k] = (A[k - 1] * A[k - 1] + k) % n;
      j = j + 1;
   }
}

for (i = n - 2 downto 0 step 1) {
   for (j = 0 to i step 1) {
      if (A[j] > A[j + 1]) {
         t = A[j];
         A[j] = A[j + 1];
         A[j + 1] = t;
      }
   }
}
```

The output for this file is given below along with commented excerpts from the input, that generate the different parts of the output. Note that you do not have to handle comments (this can be done by lex though). The comments are added to the left column for your understanding only. If time permits, separate the output chunks by empty lines.

Annotated fragments from input	Output

```
                                                  ------------------------------------------------------------
                                                  Operation       Argument 1          Argument 2          Result
                                                  ------------------------------------------------------------

                          X = 2000 // 7;     1000:  mkrtl           2000                7                   t1
                                             1001:  rstr            t1                                      x

                          y = 3000 // 8;     1002:  mkrtl           3000                8                   t2
                                             1003:  rstr            t2                                      y

                                             1004:  int2rtl         100                                     t3
                          if (x < 100 ||     1005:  jmprlt          x                   t3                  1012
                                             1006:  jmp                                                     1007
                                             1007:  int2rtl         200                                     t4
                              x > 200 &&     1008:  jmprgt          x                   t4                  1010
                                             1009:  jmp                                                     1013
                                 x != y)     1010:  jmprne          x                   y                   1012
                                             1011:  jmp                                                     1013

                              { z = 0; }     1012:  istr            0                                       z

                                             1013:  int2rtl         100                                     t5
                        if ( (x < 100 ||     1014:  jmprlt          x                   t5                  1019
                                             1015:  jmp                                                     1016
                                             1016:  int2rtl         200                                     t6
                            x > 200) &&      1017:  jmprgt          x                   t6                  1019
                                             1018:  jmp                                                     1022
                               x != y )      1019:  jmprne          x                   y                   1021
                                             1020:  jmp                                                     1022
                              { z = 1; }
                                             1021:  istr            1                                       z

                                n = 100;     1022:  istr            100                                     n

                              j = n - 1;     1023:  iadd            n                   1                   t7
                                             1024:  istr            t7                                      j

                              for (i = 0     1025:  istr            0                                       i
                                to j - 2     1026:  isub            j                   2                   t8

        /* go to loop body if i <= j - 2,    1027:  jmpile          i                   t8                  1033
                  else skip loop body */     1028:  jmp                                                     1057

             step 2.0) /* i = i + 2.0; */    1029:  flt2int         2.000000                                t9
                                        {    1030:  iadd            i                   t9                  t10
                                             1031:  istr            t10                                     i
                                             1032:  jmp                                                     1026

                            if (i == 0)      1033:  jmpieq          i                   0                   1035
                                             1034:  jmp                                                     1038
```

```
                                                  1035:   imul           i                  8                  t11
                                { A[i] = 1.;      1036:   fstrarr        1.000000           t11                A
                                                  1037:   jmp                                                   1056
                                          }
                                       else       1038:   idiv           i                  2                  t12
                                          {       1039:   istr           t12                                   k
                                  k = i / 2;
                                                  1040:   isub           k                  1                  t13
        A[k] = (A[k - 1] * A[k - 1] + k) % n;     1041:   imul           t13                8                  t14
                                                  1042:   fgetarr        A                  t14                t15
                                                  1043:   isub           k                  1                  t16
                                                  1044:   imul           t16                8                  t17
                                                  1045:   fgetarr        A                  t17                t18
                                                  1046:   fmul           t15                t18                t19
                                                  1047:   int2flt        k                                     t20
                                                  1048:   fadd           t19                t20                t21
                                                  1049:   flt2int        t21                                   t22
                                                  1050:   irem           t22                n                  t23
                                                  1051:   int2flt        t23                                   t24
                                                  1052:   imul           k                  8                  t25
                                                  1053:   fstrarr        t24                t25                A

                                                  1054:   iadd           j                  1                  t26
                                  j = j + 1;      1055:   istr           t26                                   j
                                          }       1056:   jmp                                                   1029
                                          }
                                                  1057:   isub           n                  2                  t27
                            for (i = n - 2        1058:   istr           t27                                   i

                                                  1059:   jmpige         i                  0                  1064
    downto 0 /* if i >= 0, enter outer loop,      1060:   jmp                                                   1090
                    else skip outer loop */
                                                  1061:   isub           i                  1                  t28
                    step 1) /* i = i - 1 */        1062:   istr           t28                                   i
                                                  1063:   jmp                                                   1059

                                          {       1064:   istr           0                                     j
                                  for (j = 0
                                                  1065:   jmpile         j                  i                  1070
        to i /* if j <= i, enter inner loop,      1066:   jmp                                                   1089
                    else skip inner loop */
                                                  1067:   iadd           j                  1                  t29
                    step 1) /* j = j + 1 */        1068:   istr           t29                                   j
                                                  1069:   jmp                                                   1065
                                          {
                                                  1070:   imul           j                  8                  t30
                     if (A[j] > A[j + 1]) {       1071:   fgetarr        A                  t30                t31
                                  /* A[j] */       1072:   iadd           j                  1                  t32
                                                  1073:   imul           t32                8                  t33
                                                  1074:   fgetarr        A                  t33                t34
                            /* A[j + 1] */         1075:   jmpfgt         t31                t34                1077
                                                  1076:   jmp                                                   1088

                                                  1077:   imul           j                  8                  t35
                                  t = A[j];       1078:   fgetarr        A                  t35                t36
                                                  1079:   fstr           t36                                   t

                                                  1080:   iadd           j                  1                  t37
                          A[j] = A[j + 1];        1081:   imul           t37                8                  t38
                                                  1082:   fgetarr        A                  t38                t39
                                                  1083:   imul           j                  8                  t40
                                                  1084:   fstrarr        t39                t40                A

                                                  1085:   iadd           j                  1                  t41
                            A[j + 1] = t;         1086:   imul           t41                8                  t42
                                                  1087:   fstrarr        t                  t42                A
                                          }       1088:   jmp                                                   1067
                  } /* end of inner loop */       1089:   jmp                                                   1061
                  } /* end of outer loop */       1090:
```

A note on MathC-like languages

It is a user-friendly feature to allow changes in `var` (the loop variable) and/or the variables involved in `last` and `amt`, but changing their types (in fact, the type of any variable) inside a loop is very problematic. In order to demonstrate why, look at the for loop in lines 9 – 17 of our sample program. The quads 1025 – 1056 store the translation of this loop. Just before this loop, the variable `n` gets the `int` type in line 7, and so `j` too gets the `int` type in line 8. The loop initializes `i` to the `int` value 0. Then, the code for `last` (that is, `j - 2`) is generated. Here, the code for `j - 2` involves an `int` subtraction. In this example, `amt` is a constant, and does not generate any explicit code (although, in general, `amt` may contain variables and require reevaluation).

Suppose that we change the increment of `j` in line 15 of the source code to `j = j + 1.0;` This forces `j` to be typecast to `flt`, and the `flt` sum is stored in `j` as type `flt`. This requires changing the quads 1054 and 1055 as follows.

```
1054′ :    int2flt    j                t26′
1054″ :    fadd       t26′   1.000000   t26
1055′ :    fstr       t26              j
```

This three-address code is fine in the first iteration of the loop. But the loop is reentered with `j` already having the type `flt`, so the `int2flt` conversion in the quad 1054′ no longer applies.

This is not the only problem. In quad 1056, the program makes a jump to quad 1029. In quads 1029 – 1032, there is no problem, because `i` continues to retain its `int` type. Then there is a jump to quad 1026, where `j` is subjected to an `int` subtraction. But after the first iteration of the loop, `j` is of type `flt`. At the time when the code for `j - 2` was generated, the compiler had no idea that this would be caused by a statement that it would encounter in future.

We do not see a straightforward remedy to mitigate this problem. Nevertheless, this example emphasizes the role of variable declarations along with their types, before they are used. For languages like MathC which do not require variable declarations, and which support dynamic type changes of variables, settling these (and potentially many other) issues is difficult (if possible at all) during the time of compilation. It seems that the only solutions are: (i) interpret MathC programs, and (ii) compile MathC programs to an intermediate representation that will be executed (as interpreter again) by a specially designed run-time environment.

Writing compilers is no easy task. Neither is designing programming languages. A simple user-friendliness added to a language may incur huge implications that may curtail the possibilities open even to the most consummate compiler designers.