# CSCE 531, Spring 2018, Matthews
# Project 3: Boolean Expressions and Control Flow
# Assigned: March 29, Due: April 9, 11:55PM

March 30, 2018

## 1 Overview

This assignment is an extension to Project, that provides semantic actions to generate quadruples for Booleans and Control Flow Statements using the **positional representation** implementation of booleans. This of course includes attributes B.true and B.false and auxillary functions such as "backpatch" for manipulating these lists of quadruples.

## 2 Boolean Expressions

In this addition to your compiler project we will add the intermediate code generation facilty for the boolean expressions we have discussed in class. In particular the expressions are given by the following grammar:

| | | | |
|---|---|---|---|
| $B$ | $\rightarrow$ | E relop E | relop token |
| | \| | B and B | and token, lexeme "`&&`" |
| | \| | B or B | or token, lexeme "`||`" |
| | \| | '!' B | |
| | \| | '(' B ')' | |

For this grammar you will have to also insert markers "M" for saving the value of nextquad at the appropriate locations. To support this your lexical analyzer will also need to be extended to recognize the tokens given in the table below:

| **Token** | **Lexemes** |
|---|---|
| AND | `&&` |
| OR | `\|\|` |
| NOT | ! |
| RELOP | `>, <, =, >=, <=, !=` |

If you consider the boolean `(a < b) || !(b < c) && (c < d)` how do you know what it means? That's right; the precedences of the operators, which of course you will need to ensure that your grammar will enforce.

### 2.1 Relational Operators

The "relop" token should have an attribute reltype of type int. This means you have to have a "`%union`. For each one of these statements when your parser reduces `B -> E relop E` it should generate a conditional goto of the right type followed by an unconditional goto. For example `x + y <= z` the parser should generate

| Opcode | Left | Right | Result | BranchTarget | Comment |
|---|---|---|---|---|---|
| ADD | x | y | #T57 | | |
| IFLE | #T57 | z | null | void | Put on B.true |
| GOTO | null | null | null | void | Put on B.false |

In this null is a null value where a pointer into the symbol table is expected. The value void is an integer and should eventually be replaced when we determine where it should branch. The value #T57 is the lexeme of a temporary variable. Really in the quadruple we put pointers into the symbol table for the corresponding identfiers.

In implementing the relational operators we recognize several lexemes and each has a corresponding opcode and opcode-mnemonic.

## 3 Control Flow Statements

The control flow statements are those of the core language. There are a couple of small changes.

1. For the Input and Output statements gen(IN, ID, _, _, _) and gen(OUT, ID, _, _, _) and So output x, y, z; should generate three quads, one for each ID.
2. There is a semicolon terminating every statement.
3. There is a new statement a middle exit loop that is described in the section after the grammar.

| | | | |
|---|---|---|---|
| $S$ | $\rightarrow$ | id assignOp E ';' | assignments |
| | \| | if B then S endif ';' | |
| | \| | if B then S else S endif ';' | |
| | \| | while B loop S endloop ';' | |
| | \| | loop L until B L endloop ';' | Middle exit loop |

### 3.1 Middle Exit Loop

The Middle Exit loop is shown as the last line in the grammar. It starts with the keyword "loop," then a statement list L1, then the keyword "until," then a boolean expression, another statement list L2 and finally the keyword "endloop" and the terminating ';'. The semantics of this instruction are

1. first the statementlist L1 is executed.
2. the boolean B is evaluated.
3. if B evaluates to true then the loop exits.
4. if B evaluates to false then statement list L2.
5. after L2 exists the loop should start over with statement list L1.

## 4 Intermediate Code Generation

The gen() function that generates code should be modified to generate an array of code as opposed to printing it out as you go as was done in the "Postfix" example. There also should be a global variable nextquad that keeps track of the index of the next quadruple to be generated.
To be able to support the translation assume that we have the following additional quadruples in our intermediate language.

| Opcode | Left | Right | Result | BranchTarget | Description |
|---|---|---|---|---|---|
| GOTO | | | | n | Goto quad number n |
| IFLT | left.place | right.place | | n | if left.pace < right.place then goto n |

Associated with these new opcodes you should develop a set of defines such as:

```
#define GOTO   600
#define IFLT   601
```

⋮

```
#define IFNE   ...
```

### 4.1 Dumping the Code

The codes chosen work most effectively if they are contiguous, because then one can define an array of labels and then index into the array using the opcode as "printf("%s", branchOpcode[opcode - GOTO]);".

```
char *branchOpcode[] ={''GOTO'', ''IFLT'', ... ''IFNE'' };
```

In printing the code it will be necessary to follow an explicit format.

```
Quad\# <TAB> Opcode <TAB> LeftOp <TAB> RightOp <TAB> Result <TAB> BranchTarget
```

## 5 Summary of Deliverables

You should submit a single tar-file containing your files. Specific instructions on how you should construct this are given after the list of files. The files submitted via dropbox should include at least:

1. core.y - Bison specification file for core
2. core.l - Flex specification file for core
3. Makefile with at least the targets
   (a) mycore - my core compiler
   (b) clean - remove all executables, and files that are generated, e.g., lex.yy.c
   (c) nodebug - make a non debugging version of mycore
   (d) test run mycore on your test files
4. any other necessary files, like kr.symtab.c, etc.
5. test1, $\cdots$ testn - test files for your compiler
6. out1, $\cdots$ outn - the output from testing your compiler

Note these files should placed in a directory name project3. You should then prepare a tar file with the following sequence of commands.

```
cd project3
make clean
cd ..
tar cvf proj3.tar project3
```