Project 2: Compiler Optimizations on an IR

The goal of this project is to write a compiler that translates a program of the toy language *Mini* to an intermediate representation (a CFG with IR instructions), performs some optimizations on it and finally does register allocation by graph coloring. Code generation is not part of this project. The project can either be implemented in Java or in C#.

Mini is a simple Pascal-like language with integer variables and (multi-dimensional) arrays. It has the usual kinds of statements and expressions. There are no procedures. The syntax of *Mini* is as follows:

```
Mini        = "PROGRAM" {VarDecl} "BEGIN" StatSeq "END" ".".
VarDecl     = "VAR" {IdListDecl ";"}.
IdListDecl  = ident {"," ident} ":" Type.
Type        = ident | "ARRAY" number "OF" Type.
StatSeq     = Statement {";" Statement}.
Statement   =
  [ Designator ":=" Expression
  | "IF" Condition "THEN" StatSeq {"ELSIF" Condition "THEN" StatSeq} ["ELSE" StatSeq] "END"
  | "WHILE" Condition "DO" StatSeq "END"
  | "READ" Designator
  | "WRITE" Expression
  ].
Condition   = Expression Relop Expression.
Expression  = [Addop] Term {Addop Term}.
Term        = Factor {Mulop Factor}.
Factor      = Designator | number | "(" Expression ")".
Designator  = ident {"[" Expression "]"}.
Relop       = "=" | "#" | "<" | ">" | ">=" | "<=".
Addop       = "+" | "-".
Mulop       = "*" | "/" | "%".
```

The lexical structure of SL is:

```
ident       = letter {letter | digit}.
number      = digit {digit}.
```

Integer types are written as INTEGER. Comments start with "//" and go to the end of the line.

The project consists of the following subtasks:

1. Scanner and Parser

Use the compiler generator Coco/R (http://ssw.jku.at/coco/) to generate a scanner and a parser for *Mini*. Write a small main program *Mini.java* that allocates the scanner and the parser, calls the parser and prints the number of errors that were detected. For example, you can try your compiler on the following sample program:

```
PROGRAM
VAR
    a: ARRAY 10 OF ARRAY 10 OF INTEGER;
    i, j: INTEGER;
BEGIN
    //--- read some data
    i := 0;
    WHILE i < 10 DO
        j := 0;
        WHILE j < 10 DO READ a[i][j]; j := j + 1 END;
        i := i + 1
    END;
```

```
    //--- add 1 to every value
    i := 0;
    WHILE i < 10 DO
        j := 0;
        WHILE j < 10 DO a[i][j] := a[i][j] + 1; j := j + 1 END;
        i := i + 1
    END
END.
```

2. Symbol Table Handling and Type Checking

Implement a symbol table that holds all declarations of a *Mini* programs as well as the types of the variables. Although scopes are not really necessary in *Mini*, keep a Universe scope that holds the type INTEGER. The only real scope is the one that holds the variables of the program.

Add type checking for assignments, read/write statements, and expressions. Assignments are only allowed for INTEGER values. The same holds for operands of expressions as well as for the arguments of read/write statements.

3. Intermediate Representation

Extend your compiler so that it builds a control flow graph (CFG) for the statements of the *Mini* program. The basic blocks should contain instructions similar to the ones discussed in the lecture. Every instruction has 2 operands that can reference a variable, a constant or some other instruction (see below). Some instructions have operands that are null (denoted by "-").

0	neg	x neg -	negate
1	plus	x plus y	plus
2	minus	x minus y	minus
3	times	x times y	times
4	div	x div y	divide
5	rem	x rem y	remainder
6	cmp	x cmp y	compare
7	phi	x phi opds	phi instruction
8	ld	x load y	load
9	lr	- lr y	load register y (only used after removal of phis)
10	lc	- lc y	load constant y (only used after removal of phis)
11	st	x store y	store y to the memory address denoted by x
12	ass	x ass y	assign y to the variable x
13	read	- read -	read integer value
14	write	- write y	write integer value y
15	ret	- ret -	return (used at the end of the program)
16	br	- br y	branch to block y
17	blt	x blt y	branch to block y if x references "a cmp b" and a < b
18	beq	x beq y	branch to block y if x references "a cmp b" and a == b
19	bgt	x bgt y	branch to block y if x references "a cmp b" and a > b
20	bge	x bge y	branch to block y if x references "a cmp b" and a >= b
21	bne	x bne y	branch to block y if x references "a cmp b" and a != b
22	ble	x ble y	branch to block y if x references "a cmp b" and a <= b

The instructions should be generated in SSA form as discussed in the lecture. While generating the CFG, build also its dominator tree.

You can assume that all variables reside in a single stack frame to which the special register FP points. Arrays also reside in this stack frame. Their elements are accessed as ld x, y, where x denotes a base register and y denotes either an offset or an index register.

Implement also methods that print the CFG and its instructions so that you can check their correctness. For example, for the following program:

```
PROGRAM
VAR
    i, sum: INTEGER;
    a: ARRAY 10 OF INTEGER; // adr(a) = FP-48
BEGIN
    i := 0;
    WHILE i < 10 DO READ a[i]; i := i + 1 END;
    i := 1; sum := 0;
    WHILE i < 10 DO sum := sum + (a[i] - a[i-1]); i := i + 1 END;
    WRITE sum;
END.
```

The output could be as follows:

```
1 ---------- left:3 -- right:2 -- dom:0 -- pred:
3 ---------- left:4 -- right:0 -- dom:1 -- pred: 1
   4: i := 0
4 --- while --- left:5 -- right:6 -- dom:3 -- pred: 3 5
   6: i PHI [i4 i19 ]
  16: ST ,   a
   8: i6 CMP 10
   9: (8) BGE 6
5 ---------- left:0 -- right:4 -- dom:4 -- pred: 4
  11:  RD
  12: i6 * 4
  13: FP + -48
  14: (13) + (12)
  15: ST (14), (11)  a
  18: i6 + 1
  19: i := (18)
  20: BR 4
6 ---------- left:7 -- right:0 -- dom:4 -- pred: 4
  22: i := 1
  23: sum := 0
7 --- while --- left:8 -- right:9 -- dom:6 -- pred: 6 8
  25: i PHI [i22 i41 ]
  26: sum PHI [sum23 sum39 ]
  27: i25 CMP 10
  28: (27) BGE 9
8 ---------- left:0 -- right:7 -- dom:7 -- pred: 7
  30: i25 * 4
  31: FP + -48
  32: LD (31), (30)  a
  33: i25 - 1
  34: (33) * 4
  35: FP + -48
  36: LD (35), (34)  a
  37: (32) - (36)
  38: sum26 + (37)
  39: sum := (38)
  40: i25 + 1
  41: i := (40)
  42: BR 7
9 ---------- left:2 -- right:0 -- dom:7 -- pred: 7
  44: WRT sum26
2 ---------- left:0 -- right:0 -- dom:1 -- pred: 9 1
  45: i PHI [i25 ? ]
  46: sum PHI [sum26 ? ]
  17: ST ,   a
  47: RET
```

4. Optimizations

Using the IR of the previous step, implement *copy propagation* and *global common subexpression elimination* as discussed in the lecture. This should result in the following CFG:

```
1 ---------- left:3 -- right:2 -- dom:0 -- pred:
3 ---------- left:4 -- right:0 -- dom:1 -- pred: 1
4 --- while --- left:5 -- right:6 -- dom:3 -- pred: 3 5
    6: i PHI [0 (18) ]
    8: i6 CMP 10
    9: (8) BGE 6
5 ---------- left:0 -- right:4 -- dom:4 -- pred: 4
   11:  RD
   12: i6 * 4
   13: FP + -48
   14: (13) + (12)
   15: ST (14), (11)  a
   18: i6 + 1
   20: BR 4
6 ---------- left:7 -- right:0 -- dom:4 -- pred: 4
7 --- while --- left:8 -- right:9 -- dom:6 -- pred: 6 8
   25: i PHI [1 (40) ]
   26: sum PHI [0 (38) ]
   27: i25 CMP 10
   28: (27) BGE 9
8 ---------- left:0 -- right:7 -- dom:7 -- pred: 7
   30: i25 * 4
   31: FP + -48
   32: LD (31), (30)  a
   33: i25 - 1
   34: (33) * 4
   36: LD (31), (34)  a
   37: (32) - (36)
   38: sum26 + (37)
   40: i25 + 1
   42: BR 7
9 ---------- left:2 -- right:0 -- dom:7 -- pred: 7
   44: WRT sum26
2 ---------- left:0 -- right:0 -- dom:1 -- pred: 9 1
   45: i PHI [i25 ? ]
   46: sum PHI [sum26 ? ]
   47: RET
```

5. Register Allocation

This step is optional. Do it if you have enough time and if you are interested in the details of register allocation implementation.

Use the algorithm shown in the lecture to build an interference graph. While traversing the CFG you should also eliminate unused instructions. Color the graph assuming that you have 30 registers available (2 registers are reserved as scratch registers).

After graph coloring try to coalesce the operands of phi instructions so that the phi instructions become unnecessary. Then remove the phi instructions, possibly inserting register moves (i.e., lr or lc instructions) at the end of the predecessor blocks.

Finally print out the CFG using the assigned registers for the operands of the instructions. The result should look as follows:

Output	*Corresponds to*
1 ----------	
3 ----------	
R3: LC 0	R3 = 0
4 ----------	
R0: R3 CMP 10	4: R0 = R3 CMP 10
: R0 BGE 6	BGE R0, 6
5 ----------	
R2: RD	R2 = RD
R0: R3 * 4	R0 = R3 * 4
R1: FP + -48	R1 = FP - 48
R0: R1 + R0	R0 = R1 + R0
: ST R0, R2	mem[R0] = R2
R3: R3 + 1	R3 = R3 + 1
: BR 4	BR 4
6 ----------	
R3: LC 1	6: R3 = 1
R4: LC 0	R4 = 0
7 ----------	
R0: R3 CMP 10	7: R0 = R3 CMP 10
: R0 BGE 9	BGE R0, 9
8 ----------	
R0: R3 * 4	R0 = R3 * 4
R1: FP + -48	R1 = FP - 48
R2: LD R1, R0	R2 = mem[R1 + R0]
R0: R3 - 1	R0 = R3 - 1
R0: R0 * 4	R0 = R0 * 4
R0: LD R1, R0	R0 = mem[R1 + R0]
R0: R2 - R0	R0 = R2 - R0
R4: R4 + R0	R4 = R4 + R0
R3: R3 + 1	R3 = R3 + 1
: BR 7	BR 7
9 ----------	
: WRT R4	9: WRT R4
2 ----------	
: RET	RET