CS406: Compilers

Maximum Points: 8 Written Assignment 2 Due: 23/3/2021, 5PM

Instructions:
The assignment is individual and paper-based. State your assumptions (if any) clearly. You must turn-in your handwritten assignment sheets to the instructor or TAs before the due date and time.

1. **(1 point)** consider the following grammar rule for `if_stmt` with associated action routines. Assume that a helper function `new_label()` returns a new label (`string` type) every time it is called.

```
<if_stmt> -> if {do=start_if();} <b_expr> {testcond_if(do);} then
<stmt_list>
   { elsif {gen_jump(do); gen_else_label();} <b_expr> {testcond_if();} then
<stmts> }
   <else_part> endif; {gen_out_label();}

   <else_part> -> else {gen_jump(do); gen_else_label();} <stmt_list>
   <else_part> -> {gen_else_label();}
```

The semantic record for if statement:

```
struct if_stmt {
      string out_label, next_else_label;
};
```

The action routines are below:
`//start_if()` calls `new_label()` that creates a label that is the target of all statements out of the if statement.

```
if_stmt start_if(void) {
      if_stmt do = new if_stmt();
      do.out_label = new_label();
      do.next_else_label = " ";
      return do;
}
```

//the `next_else_label` is initialized to blank above because all else labels are created in the `testcond_if` routine, which generates a conditional jump over the following **`then`** part

```
void testcond_if(if_stmt& do) {
      check if b_expr.data_object.object_type == BOOLEAN
            do.next_else_label = new_label();
            //JUMP0 is jump if false
            generate(JUMP0, <b_expr>.data_object, do.next_else_label," ");
}
```

`//gen_jump` is the action routine called to generate jump tp the `out_label` if an `else` or `elsif` part follows a `then` part.

```
gen_jump(if_stmt& do){
      generate(JUMP, do.out_label, " ", " ");
```

```
}
```

//gen_else_label() labels the beginning of an else or elsif part with the label generated by testcond_if() as the target of the last conditional jump.

```
gen_else_label(if_stmt& do){
    generate(LABEL, do.next_else_label, " ", " ");
}
```

//After the entire statement has been processed, gen_out_label() generates the LABEL tuple for the out_label created by start_if()

```
gen_out_label(if_stmt& do){
    generate(LABEL, do.out_label, " ", " ");
}
```

If you were implementing a one-pass compiler to generate binary code directly for the if_stmt, why would the above routines not work? Your explanation/answer must refer to the action routines above (0.5 points). How would you make it work (0.5 points)?

2. **(4 points)** Assume that the following program is running on a machine with 4 registers (32-bit), using callee saves. Assume addresses (i.e., pointers) are 8 bytes, floats are 4 bytes, ints are 4 bytes, and doubles are 8 bytes. Draw the complete stack (i.e., the stack including all active activation records) for the program right after *foo has called bar, and before bar returns*. For each slot in the stack, indicate what is stored there, and how much space that slot takes up.

```
void main() {
    int x;
    double y;
    float f;
    ... //some computations
    f = foo(x + y);
    f += bar(x, y);
    ... //some computations
}

float foo(int a) {
    double g;
    g = bar(a, a);
    return g;
}

double bar(int r, int s) {
    float h;
    h = 1.0 * (r + s);
    if (r == s) {
```

```
            double q = 2.0;
            h = h * q;
        }
        return h;
    }
```

3. **(3 points)** Perform bottom-up register allocation on the code for a machine with three registers. Show what code would be generated for each 3AC instruction. When choosing registers to allocate, always allocate the lowest-numbered register available. When choosing registers to spill, choose the non-dirty register that will be used farthest in the future. In case all registers are dirty, choose the register that will be used farthest in the future. In case of a tie, choose the lowest-numbered register.

```
1: A = 7
2: B = A + 2
3: C = A + B
4: D = A + B
5: A = C + B
6: B = C + B
7: E = C + D
8: F = C + D
9: G = A + B
10: H = E + F
11: I = H + G
12: WRITE(I)
```