Problem 1

Recall the type checker based on an attribute grammar that we discussed in lectures 13/14. For an additional reference, see Sethi Ch. 13.2 and/or ASU Ch. 5.1 on attribute grammars.

Write an attribute grammar that implements a compiler for a simple expression language for a RISC (reduced instruction set) target architecture. Your attribute grammar has to perform type checking, and generate code using the results of your type checker. The following sections describe the (1) expression language, (2) type system, and (3) the specification of the target architecture. You may receive partial credit if you only do the type checker.

1. Expression language in BNF

   `<program> ::= <stmt_list>
   <stmt_list> ::= <stmt> ; <stmt_list> | <stmt>
   <stmt> ::= id := <expr> | id := intnum | id := floatnum | print(id)
   <expr> ::= id + id | id * id | id / id`

   `id` is an alpha-numerical name (e.g.: `a1, bb`), `intnum` is an integer number (e.g.: `4, 55`), and `floatnum` is a floating point number (e.g.: `3.14, 44.01`).

2. Type system

   Each language construct has a type. Statements can have type `void` or `type_error`, and expressions can have type `integer`, `float`, or `type_error`. Identifiers can have type `integer`, `float`, or `type_error`. Integer constant can only have type `integer`, while floating point constants can only have type `float`. Identifiers don’t have to be explicitly declared in our language, i.e., the language is implicitly typed. The type of a variable can be determined from the way it is used. Each variable has a unique type and this type will be determined by the first assignment to the variable. The following table gives the type rules for the operations in our language.

   + | overloaded; implicit coercion | `+ : integer × integer → integer`
   | | `+ : float × float → float`
   * | implicit coercion | `* : float × float → float`
   / | no coercion | `/ : float × float → float`
   ::= | overloaded; no coercion | `:= : integer × integer → void`
   | | `:= : float × float → void`
   print | overloaded; no coercion | `print : integer → void`
   | | `print : float → void`
If an expression contains a variable with an unknown type, i.e., its type has not yet been determined, the type of the expression will be `type error`. Note that the first assignment to a variable determines its type.

3. **Target architecture**

The target architecture is a simple RISC machine. There are four registers, two integer registers $R_a$ and $R_b$, and two floating point registers $R_x$ and $R_y$. Integer values have to be stored in integer registers, and floating point values have to be stored in floating point registers. A RISC architecture is a load/store architecture where arithmetic instructions operate on registers rather than memory operands (memory addresses). This means that for each access to a memory location, a `load` or `store` instruction has to be generated. Integer instructions have to use integer registers, and floating point instructions have to use floating point registers. Here is the machine instruction set of our RISC target architecture. $A$ and $B$ represent registers of the appropriate types.

<table>
<thead>
<tr>
<th>instr, format</th>
<th>description</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDCint A intnum</td>
<td>load integer constant</td>
<td>$A \leftarrow \text{intnum}$</td>
</tr>
<tr>
<td>LDCflt A floatnum</td>
<td>load floating point constant</td>
<td>$A \leftarrow \text{floatnum}$</td>
</tr>
<tr>
<td>LDint A id</td>
<td>load integer value of id</td>
<td>$A \leftarrow \text{id}$</td>
</tr>
<tr>
<td>LDflt A id</td>
<td>load floating point value of id</td>
<td>$A \leftarrow \text{id}$</td>
</tr>
<tr>
<td>STint id A</td>
<td>store integer value into id</td>
<td>$\text{id} \leftarrow A$</td>
</tr>
<tr>
<td>STflt id A</td>
<td>store floating point value into id</td>
<td>$\text{id} \leftarrow A$</td>
</tr>
</tbody>
</table>

**arithmetic instructions**

<table>
<thead>
<tr>
<th>instr, format</th>
<th>description</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDint A B</td>
<td>add contents of two registers and store result in first register</td>
<td>$A \leftarrow A + B$</td>
</tr>
<tr>
<td>ADDflt A B</td>
<td>add contents of two registers and store result in first register</td>
<td>$A \leftarrow A + B$</td>
</tr>
<tr>
<td>MULint A B</td>
<td>multiply contents of two registers and store result in first register</td>
<td>$A \leftarrow A \times B$</td>
</tr>
<tr>
<td>MULflt A B</td>
<td>multiply contents of two registers and store result in first register</td>
<td>$A \leftarrow A \times B$</td>
</tr>
<tr>
<td>DIVint A B</td>
<td>divide contents of two registers and store result in first register</td>
<td>$A \leftarrow A / B$</td>
</tr>
<tr>
<td>DIVflt A B</td>
<td>divide contents of two registers and store result in first register</td>
<td>$A \leftarrow A / B$</td>
</tr>
</tbody>
</table>

**other instructions**

<table>
<thead>
<tr>
<th>instr, format</th>
<th>description</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTIF A B</td>
<td>convert integer value in B into a floating point value, and store in floating point register A</td>
<td>$A \leftarrow \text{(float)} \ B$</td>
</tr>
<tr>
<td>PRT A</td>
<td>print contents of register A</td>
<td>print($A$)</td>
</tr>
<tr>
<td>STOP</td>
<td>end of program execution</td>
<td>stop</td>
</tr>
</tbody>
</table>
Your attribute grammar can assume that

- token id has an attribute name that contains the actual character sequence of the identifier.
- token intnum has an attribute value that contains the actual integer value for the constant
- token floatnum has an attribute value that contains the actual floating point value for the constant

In addition, you have to use two more attributes for expressions and statements: type and code. Your attribute grammar should return the entire code sequence for the program in <program>.code, i.e., the code attribute of the root of the parse tree. Here is the basic structure of your attribute grammar:

```
<program> ::= <stmt_list>
            {<program>.type ← <stmt_list>.type;
             if <program>.type = void then
             <program>.code ← <stmt_list>.code || "STOP"}

<stmt_list> ::= <stmt> ;<stmt_list> { ... }
<stmt> ::= <stmt>
         { ... }
<stmt> ::= id := <expr>
         { ... }
<stmt> ::= id := intnum
         { ... }
<stmt> ::= id := floatnum
         { ... }
<stmt> ::= print( id )
         { ... }
<expr> ::= id + id
         { ... }
<expr> ::= id * id
         { ... }
<expr> ::= id / id
         { ... }
```

The operator || represents concatenation of machine instructions. Note that if the program contains a type error, <program>.code is empty, i.e., remains undefined.

Here is an example of a program and the code that you should generate. Note that the code is very simple, and for each statement in our input expression language, code is generated that uses at most two registers.
D := 2;
E := 3;
A := D + E;
B := 2.5;
C := B * A;
print(C)

LDClnt \( R_a \) 2
STint \( D \) \( R_a \)
LDClnt \( R_a \) 3
STint \( E \) \( R_a \)
LDInt \( R_a \) \( D \)
LDInt \( R_b \) \( E \)
ADDInt \( R_a \) \( R_b \)
STInt \( A \) \( R_a \)
LDClflt \( R_x \) 2.5
STflt \( B \) \( R_x \)
LDflt \( R_x \) \( B \)
LDInt \( R_a \) \( A \)
CVTIF \( R_y \) \( R_a \)
MULflt \( R_x \) \( R_y \)
STflt \( C \) \( R_x \)
LDflt \( R_x \) \( C \)
PRI \( R_x \)
STOP

Problem 2

type foo = record
  pressure : array(40,20,100) of real;
  angle : integer;
  speed : double;
  end;

foo-bar : foo;

... foo-bar.angle = ...
foo-bar.pressure(5, 3, 78) = ...
foo-bar.pressure(2, 19, 1) = ...
foo-bar.speed = ...

1. Give the layout of the data object of type foo that specifies the location of each of its components relative to the base (starting) address of a foo object. All addresses are word addresses. double requires two words, while real and integer have single word representations. Assume that the array references for each dimension start with 0, i.e., the range of possible indices in our example is \([0..39,0..19,0..99]\). Give a layout address function of the three-dimensional array that is an extension of the row-wise layout for two-dimensional arrays.

2. Give the relative addresses for the four component accesses of foo-bar shown above.